MOBILE DESIGN SEMANTICS

BY

THOMAS LIU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Assistant Professor Ranjitha Kumar

ABSTRACT

Given the growing number of mobile apps and their increasing impact on modern life, researchers have developed black-box approaches to mine mobile app design and interaction data. Although the data captured during *interaction mining* is descriptive, it does not expose the *design semantics* of UIs: what elements on the screen *mean* and how they are used. This thesis introduces an automatic approach for semantically annotating the elements comprising a UI given the data captured during interaction mining. Through an iterative open coding of 73k UI elements and 720 screens, we first created a lexical database of 24 types of UI components, 197 text button concepts, and 135 icon classes shared across apps. Using the labeled data created during this process, we learned code-based patterns to detect components, and trained a convolutional neural network which distinguishes between 99 icon classes with 94% accuracy. With this automated approach, we computed semantic annotations for the 72k unique UIs comprising the Rico dataset, assigning labels for 78% of the total visible, non-redundant elements.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to first thank my advisor, Dr. Ranjitha Kumar, for her invaluable assistance, mentoring, and support throughout my graduate student career here. I would also like to thank Dr. Biplab Deka, Mark Craft, Jason Situ, and Oliver Melvin for their help in my work. Finally, I am grateful to my family for their help, love, and support, without which I would not have survived the struggle of graduate school.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The ubiquity of mobile apps in everyday life and their availability in centralized app repositories make them an attractive source for mining digital design knowledge [1, 2]. Recently, Deka et al. introduced *interaction mining*, a black-box approach for capturing design and interaction data while an Android app is being used [3]. The data captured during interaction mining exposes a UI's screenshot; the elements it comprises and their render-time properties (i.e., view hierarchy); and the interactions performed on the screen and how they connect to other UI states in the app. This data provides a complete specification of a UI sufficient to reverse engineer it, but it fails to expose the *semantics* of UIs: what elements on the screen *mean* and how they are used.

This thesis presents an automated approach for semantically annotating the elements of a mobile UI, given a screenshot and view hierarchy (Figure 1.1). These annotations identify both the *structural* roles (e.g., image content, bottom navigation) and the *functional* ones (e.g., login button, share icon) that elements play in the UI design. To develop this approach, we first generated a lexical database of the UI components and UX concepts (i.e., text buttons and icons) that are shared across apps through an iterative open coding of 73k UI elements and 720 screens. Then, we leveraged this database to learn code-based patterns to detect different components, and trained a convolutional neural network (CNN) to distinguish between icon classes.

We bootstrapped the lexical database by referencing popular design libraries and app-prototyping tools to create a vocabulary of UI components and UX concepts. We refined and augmented this vocabulary through unsupervised clustering and iterative coding of more than 73k elements sampled from the Rico dataset, which comprises interaction mining data from 9.7k Android apps [4]. Through this process, we identified 24 types of UI components, 197 text button concepts, and 135 icon classes. The resultant database also exposes icon and text-button synonym sets related to each UX concept, creating links between the visual and textual elements used in digital design.

To semantically annotate mobile UIs, we employ a code- and vision-based approach that leverages the labeled data generated in the creation of the lexical database. We identified a set of textual and structural patterns in the view hierarchies that were predictive of each component class, except for icons. Icons are difficult to automatically distinguish in the code from images; therefore, we trained a convolutional neural network (CNN) which distinguishes between 99 icon classes with 94% accuracy. We pass the activation vectors produced by the CNN through a Gaussian mixture model trained to detect *anomalies*, image inputs that do not belong to any of the 99 classes. The CNN with anomaly detection differentiates between images and icons, and predicts the correct icon class 90% of the time.

With this automated approach, we computed semantic annotations for the 72k unique UIs in the

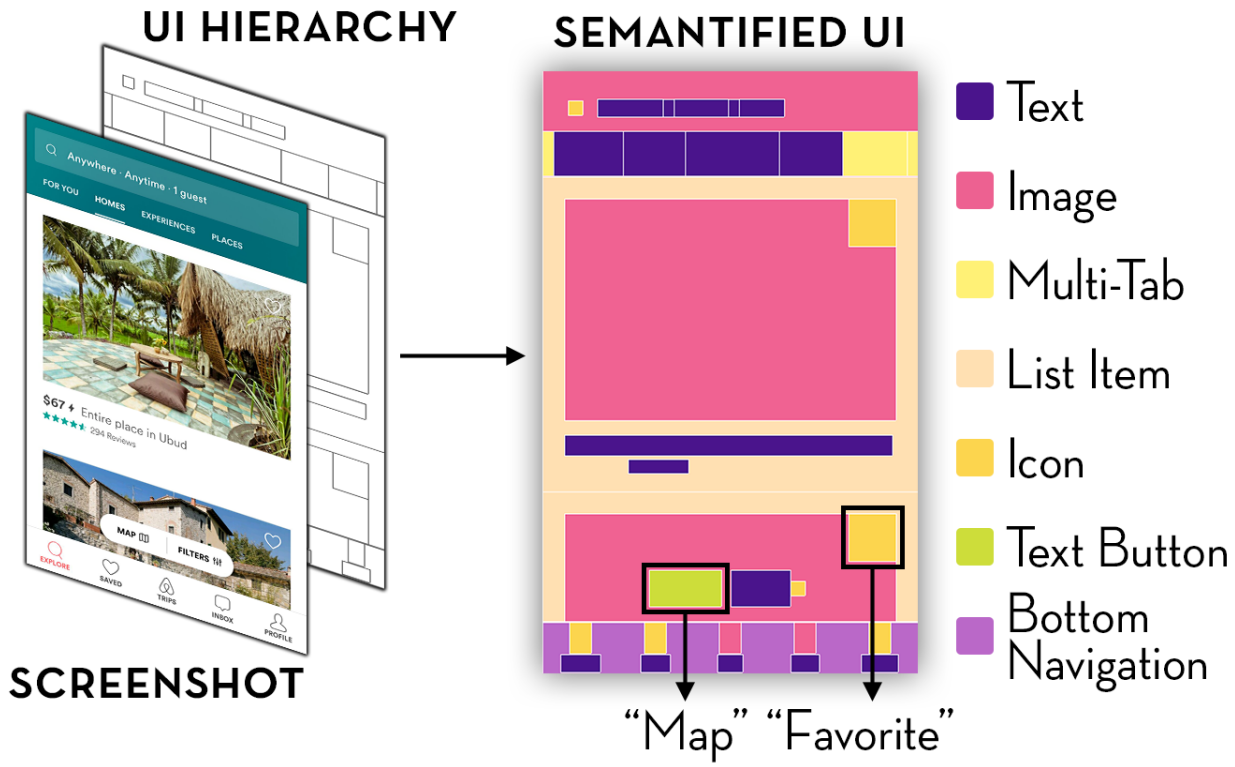

Figure 1.1: This thesis introduces a code- and vision-based approach for semantically annotating the elements comprising a mobile UI. Given its screenshot and view hierachy, we automatically identify 24 UI component categories, 197 text button concepts, and 99 icon classes shared across apps.

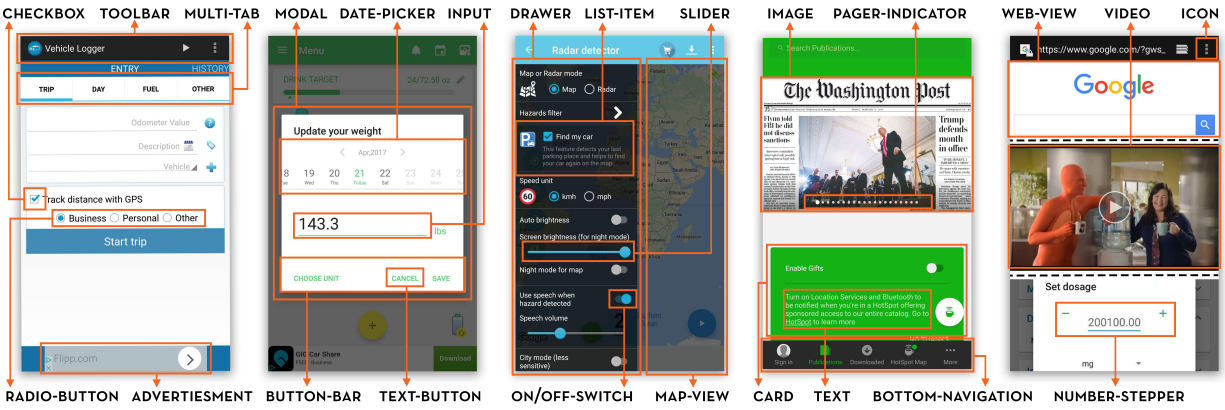

Figure 1.2: Screenshots illustrating the complete set of 24 UI component categories identified through an open iterative coding of 720 UI screens from the Rico dataset.

Rico dataset. After removing hidden and redundant elements from the view hierarchies, we achieve 78% coverage over the total number of elements in the dataset. Finally, we sketch how design semantics can power new types of search interactions, aggregation techniques for understanding patterns, and tools for generating designs.

CHAPTER 2: RELATED WORK

The systematic study of mobile UIs is not a sparse field. Many of the current collections of user interface examples exist, but are usually hand curated and static, containing little more than a few screenshots [5, 6]. The Rico mobile app dataset, a dataset comprising over 9,000 apps and 72,000 unique UI screens, mined from crowdsourced and automated UI exploration is the largest dataset of its kind, and includes textual, structural, interactive, and graphical properties of every screen, enabling a detailed understanding of every part of the screen [4]. The dataset was generated by recording the crowdsourced interactions of users with real world mobile apps downloaded from the Google Play Store, allowing for in-depth exploration of app interfaces. The dataset can be viewed at `https://rico.interactionmining.org` and downloaded at `http://interactionmining.org/rico#quick-downloads`.

Prior work has shown that Android "widgets" tend to be constructed in regular ways as groupings of Android elements [2, 1]. Shirazi et al. perform a static analysis of the XML files that define Android interfaces and exploited parent-child-element combinations to identify a common set of widgets [2]. For example, they identify two TextView elements within a LinearLayout as a common widget; however, they do not attribute any semantic meanings to these code patterns. Similarly, Alharbi et al. also identify a small set of common UI components and describe the methods that Android developers take to construct them [1].

Previous work has also been done on using annotations to cluster and label parts of a whole. Yi et al. utilized annotations on 3D models to extract labeled, geometrically segmented parts [7]. They take, for example, a 3D model of a car, and then break it down into parts such as dashboard, door, or trunk, through creating an embedding of parts, and then using expectation maximization to bring parts with similar 3D modeler labeled annotations closer together.

Classification of small images also has a lengthy history. MNIST [8], for example, is a set of handwritten digits commonly used for the training and evaluation of computer vision models. The CIFAR-100 dataset consists of 32×32 images of animals, plants, vehicles, devices, and other real life objects [9]. Most recently, exponential linear units were leveraged to obtain the best accuracy to date, 75%, on the CIFAR-100 dataset [10].

The creation of semantic databases that map different types of concepts has also been done in other domains. ImageNet [11] used WordNet as a basis to organize the large number of images present in the database. WordNet itself exposed relationships among different words, by demonstrating links such as synonyms and hypernyms [12]. These influential datasets have shown that databases designed with semantics in mind can prove enormously useful.

Further, researchers have demonstrated the automated analysis and generation of mobile UIs in

the past. For example, Beltramelli shows that prototype UIs can be generated from screenshots [13]. However, this approach seems to only work for simple UIs, and does not exploit the view hierarchy of the Android application to achieve its purpose. REMAUI allowed for the the reverse engineering of mobile app screenshots or the conversion of high quality mockups into UI code [14]. P2A further extended REMAUI by converting screen transitions as well [15]. REDRAW is yet another system for converting UI code generation tool that attempts to reproduce UI code from screenshots [16]. However, these tools are all limited by relying on computer vision for the detection of UI components and their bounds, and do not apply a semantic meaning to the components.

CHAPTER 3: CREATING A LEXICAL UI/UX DATABASE

The first contribution of this thesis is a lexical database that enumerates a comprehensive set of UI component categories and UX concepts found in mobile apps. We bootstrapped the UI/UX vocabulary by referencing popular design libraries and tools. We then augmented and refined this vocabulary through unsupervised clustering and an iterative open coding of 720 UI screens and 73k elements sampled from the Rico dataset, comprising interaction mining data from 9.7k Android apps across 27 app categories [4].

We categorized UI components to provide semantics that describe the *structural* roles (e.g., image content, bottom navigation) elements play on a screen. Similarly, we analyzed text buttons and identified icon classes to determine *functional* semantics, UX concepts that describe how elements and screens are used. This vocabulary taken together with the labeled data form the lexical UI/UX database, which we leverage to construct an automatic approach to semantically annotate mobile UIs.

3.1 RICO WEB APP

To enable the study and exploration of the Rico dataset, we designed a web application to facilitate its viewing, christened "Rico Explorer". This web app is available for public use at `http://rico.interactionmining.org/`.

3.1.1 Backend

The backend of the Rico Explorer application consists of a Flask server and a MongoDB database. Using a python script, we iterated through the entirety of the raw data of the Rico dataset, and then inserted the data into MongoDB [17]. We indexed certain fields, such as category, to allow for for fast searching over the dataset. The Flask server is largely just a RESTful wrapper around the database, though we did implement a few custom endpoints for calculating the number of results for each query, since the Flask mongorest library we used was missing that functionality [18].

3.1.2 Frontend

The frontend of the Rico Explorer application is a single page application (SPA) written in React.js [19]. The main home page view, as seen in Figure 3.1, provides an infinitely scrolling list

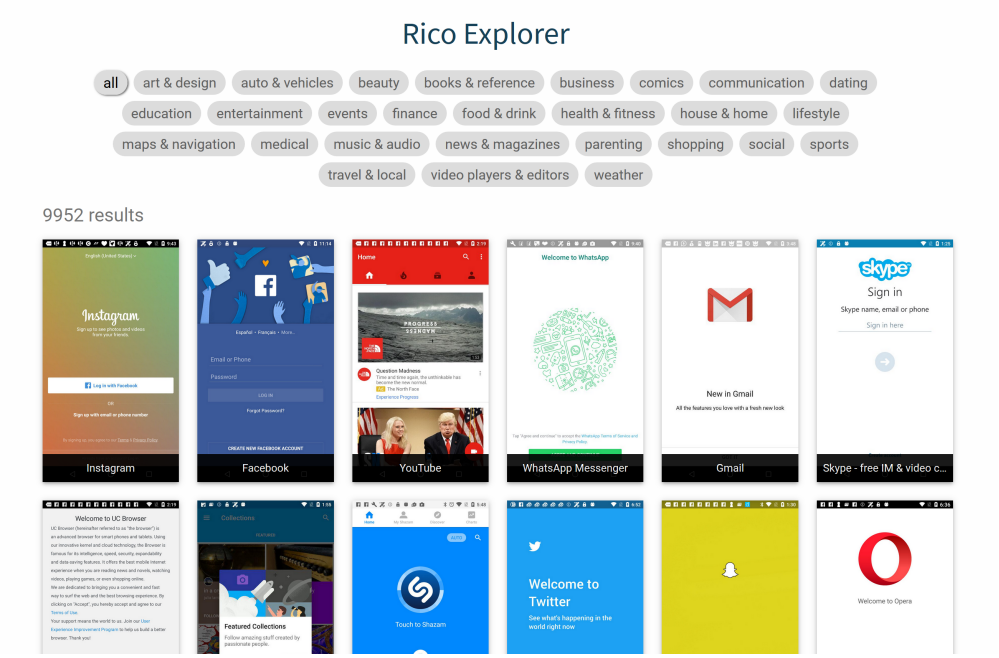

Figure 3.1: The home page of Rico Explorer, showing category selection and app results.

of all the apps within the Rico dataset. Rows of category chips along the top allow users to select the category with which they would like to filter the results.

Clicking on an app brings users to the app view, displayed in Figure 3.2, allowing them to scroll through a series of UIs that represent the recording of a user's chronological interactions with the app in question. A collection of such a series of UIs is known as a "trace". The user can also see information about the app at the top, as well as swap between traces with the dropdown menu. Additionally, keyboard shortcuts are provided for swift navigation of the traces.

Finally, clicking on a UI within a trace will bring up the detailed UI view, as seen in Figure 3.3. Users can continue scrolling through UIs, but also examine the view hierarchy in the right pane, which mirrors the UI. By hovering over elements in the view hierarchy, users can highlight the corresponding elements on the UI image on the left. Clicking on the description icon on view hierarchy elements brings up some basic information on that element, such as class name, ancestors, text, and whether the element is clickable. Users can also drill deeper into the view hierarchy to examine deeper elements. The JSON tab allows users to view the raw view hierarchy in JSON format, exposing the full amount of data that Rico provides.

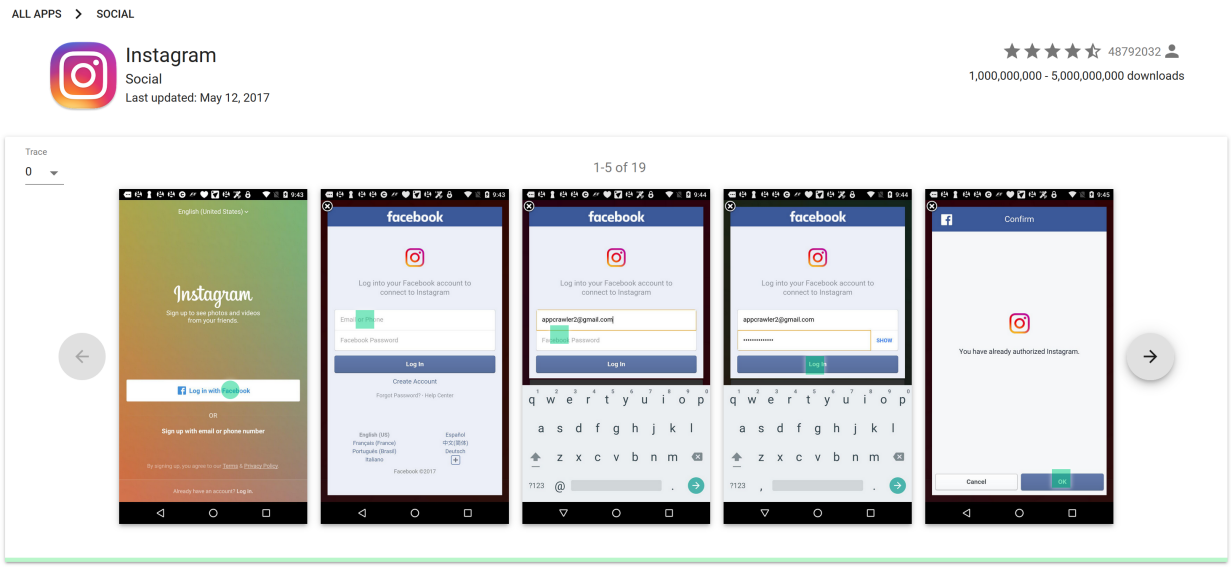

Figure 3.2: The app view of Rico Explorer, showing an overview of the trace, as well as detailed app information

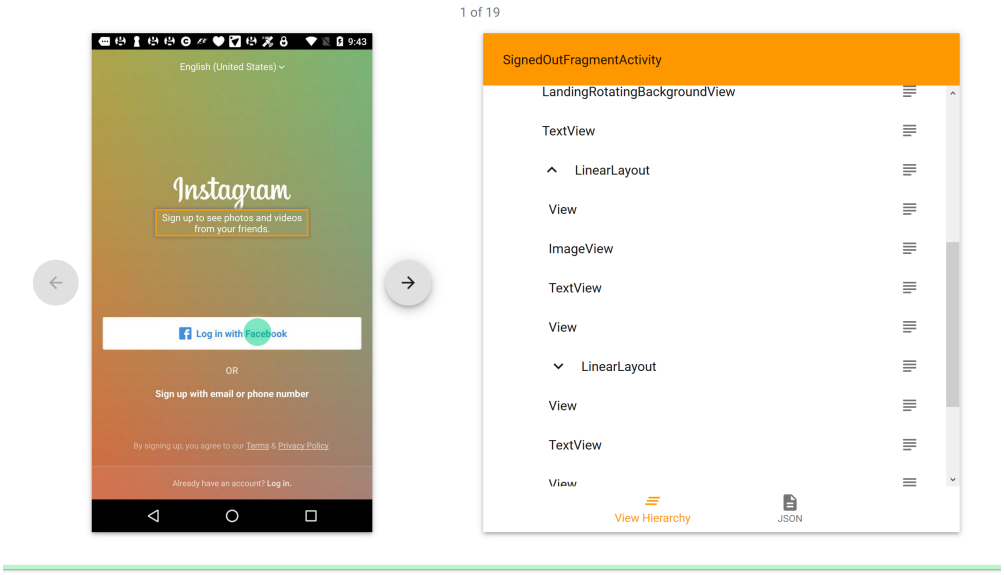

Figure 3.3: The UI view of Rico Explorer, showing the UI on the left, and the view hierarchy tree on the right. Users can drill up or down on the view hierarchy as they choose, and the corresponding UI element will be highlighted on the left

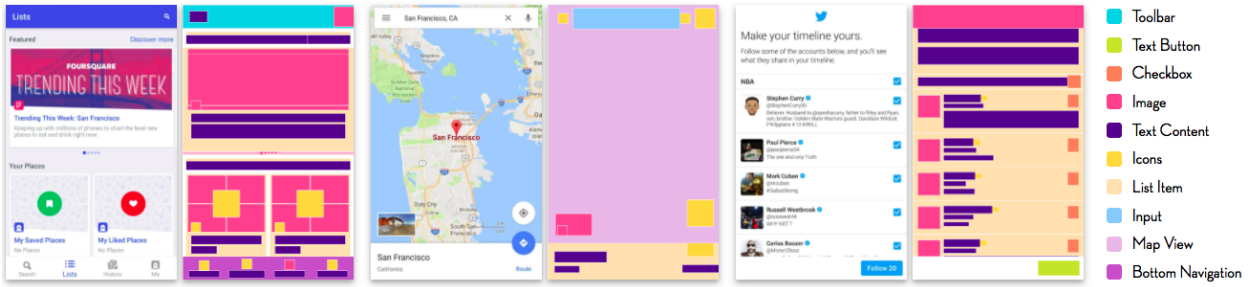

Figure 3.4: Our automated semantic approach applied to different mobile UIs. We extracted code-based patterns to semantically classify 23 out of the 24 types of UI components on a screen. To identify icons, we trained a convolutional neural network and anomaly detection pipeline which distinguishes between images and 99 common classes of icons.

3.2 UI COMPONENTS

To identify a comprehensive set of UI component categories comprising Android apps, we first referenced popular design tools and languages that expose component libraries such as Balsamiq [20] and Google's Material Design [21]. Using a consensus-driven, iterative approach, three researchers examined the component categories enumerated by these references and merged together any classes that were very similar. For example, "Date Picker" and "Date Chooser" were combined into a "Date Picker" category. Similarly, components that were determined to be subtypes of other components were merged into their parent class. For example, Material Design's "Chip" category was merged into the "Text Button" class.

To ensure that the component vocabulary was comprehensive, we also did an iterative open coding of 720 unique screens from Rico, approximately 1% of the dataset. We created this UI coding set by randomly sampling 30 apps from 24 app categories in Rico, and then randomly sampling one screen from each app. Three researchers from our team independently coded the elements comprising the 720 UI screens, noting any component types that were not part of the initial vocabulary set. After the initial coding, they met and discussed discrepancies and new component categories until agreement was reached.

After completing this iterative coding process, our final list had 24 components: Advertisement, Bottom Navigation, Button Bar, Card, Checkbox, Date Picker, Drawer, Icon, Image, Input, List Item, Map View, Modal, Multi-Tab, Number Stepper, On/Off Switch, Pager Indicator, Radio Button, Slider, Text, Text Button, Toolbar, Video, and Web View. We provide a representative example for each component in Figure 1.2. We further analyze and categorize "Text Button" and "Icon" elements to mine UX concepts.

CONCEPT (#)	BUTTON TEXT STRINGS
no (5636):	no, no thanks, decline, disagree, reject, deny, refuse
login (5632):	login, sign in, log in, sign in with google, sign in with facebook, log in with facebook, log in with google, connect with facebook, login with facebook, log in with email, log in with twitter, sign in with email
back (5479):	back, cancel, return
go (5425):	go, start, begin, get started, proceed, start now
ok (5069):	ok, got it, okay, ok got it
all (4309):	all, view all, see all, clear all, select all
next (3308):	next
add (2672):	add, add to cart, add location, add to list, add to bag, add card
create (2585):	new, create, create account, new message, create an account, create new account
more (2544):	more, learn more, more info, load more, more apps, read more, more options
retry (2502):	retry, try again
skip (2428):	skip
set (2298):	set, set wallpaper, set up my news, set language, set as home
delete (2152):	delete, remove, clear, remove ads, clear all
facebook (1840):	facebook, sign in with facebook, sign up with facebook, login with facebook, log in with facebook, connect with facebook, continue with facebook
view (1766):	view, view all
book (1720):	book, book now
continue (1720):	continue, continue as guest, continue with facebook, continue shopping
list (1558):	list, add to list, mode list
save (1548):	save, save changes, save search
search (1407):	search, save search, search flights
finish (1342):	finish, done
agree (1249):	agree, yes, i agree, confirm
show (1233):	show, show password
share (1143):	share, share location
buy (1033):	buy, buy now, buy tickets
update (991):	update, upgrade, upgrade now, update now
edit (976):	edit, customize, refine, edit profile

Figure 3.5: The 28 most frequent UX concepts mined from buttons, identified through clustering 20,386 unique button text strings.

3.3 UX CONCEPTS

To construct a set of *functional* semantics, UX concepts that describe how elements and screens are used, we analyze text buttons and icons, since users often interact with both types of components to perform different tasks (e.g., login button, search icon) [22]. We can mine UX concepts directly from the text buttons by extracting and clustering the text contained within them. To mine UX concepts from icons, we first identify classes of visually similar icons through an iterative coding of 73k elements. Then we extract words (i.e., concepts) relevant to each visual class by analyzing developer defined code properties of its examples.

3.3.1 Text Buttons

Based on the text button examples that we identified while categorizing UI components, we detect a pattern for reliably extracting text buttons from Android view hierarchies: any element whose Android class contains the string "button," or any element whose Android class inherits from a class that contains the string "button." This criteria can be verified for an element by examining its `classname` and `ancestors` properties, respectively. Using this criteria, we extracted 130,761 buttons from the Rico dataset, and found 20,386 unique button text strings.

To mine UX concepts from these buttons, we filtered out those whose text comprise a single character or appear in fewer than five apps, reducing the set to 347 unique text strings. Then we grouped together text strings that have substrings in common such as "retry" and "try again." We determine a UX concept (e.g., create, log in) for each cluster based on the most frequently occurring words in the set. Each unique text string can belong to more than one cluster. For example, "login with facebook" is part of both "login" and "facebook" concepts.

The set of concept categories and button text mappings were verified by the entire research team in a group setting. Decisions to split or merge concept groups were discussed until consensus was reached with all team members. At the end of this process, 197 text button concepts were identified. Some of these concepts and related text button strings are shown in Figure 3.5.

3.3.2 Icons

To mine UX concepts from icons and categorize them, we need to first extract a large set of icons from the Rico dataset. The icon examples identified while categorizing UI components reveal that icons are often represented in the view hierarchy as clickable images, and are visually small and squarish. Since we cannot distinguish between different classes of icons without looking at them, we need to ensure that they are visible in the screenshot.

Therefore, to identify a potential icon we used the following criteria: an image component that is `clickable` and `visible-to-user`, and whose area is less than 5% of the total screen area, and aspect ratio is greater than .75 (where aspect ratio is calculated as the smaller dimension divided by the larger dimension). We use an element's `bounds` properties to compute its area and aspect ratio, and to also automatically crop the element from the screenshot.

Using this criteria and after removing duplicate components from the same app, we create a set of 73,449 potential icons. Through an iterative open coding of this set, we identified 135 icon classes that were commonly shared across apps in the Rico dataset.

To facilitate high-quality, fast coding over a large dataset, we built a web application that sequentially showed researchers unlabeled potential icons, along with the current lexicon of categories for reference, which we bootstrapped with Google's Material icon set [23]. If the shown element fell into an existing icon category, researchers could quickly assign a label using an autocomplete input field. Through this interface, researchers could also create new categories if a relevant one did not already exist, or label an element 'not an icon.'

Once every potential icon in the 73k set had been assigned a label, the research team used another web application to review all the categories and the elements assigned to each category, and through consensus refine categories and reassign any elements that had been mislabeled. As part of this process, we had to identify distinguishing features for visually similar icons to remove ambiguity. For example, since *skip_next* and *av_forward* are visually similar, we require that *skip_next* icons have a vertical line in front of the arrows to belong to the category. Overall, 78% of the potential icons belong to 135 common classes, 13,108 icons are semantic, but too specific or niche to be further classified, and the remaining 11,205 images were determined not to be icons at all.

To map icons to UX concepts, we mine the set of words used by developers to characterize each icon class. While icons are concise, the same visual representation can have different meaning depending on the context (i.e., *polysemy*), or have a meaning that may not be immediately apparent [24]. To understand an icon category's range of meaning, we extracted the `resource-id` property associated with the examples for that class. We split each `resource-id` on underscores or via camel case parsing, and performed tf-idf to identify the substrings that are characteristic of each class. We use these words sets, which are similar to the synonym sets exposed by WordNet [12], to map icons to text concepts (Figure 3.6). Our analysis reveals not only what developers call each icon, but also how the icons are used. For example, *national_flag* is used for both location and language purposes.

Based on the categorization of UI components, text buttons, and icons, and the labeled data generated through our process, we create a lexical UI/UX database which enumerates a comprehensive set of design semantics and maps them to implementations in existing apps.

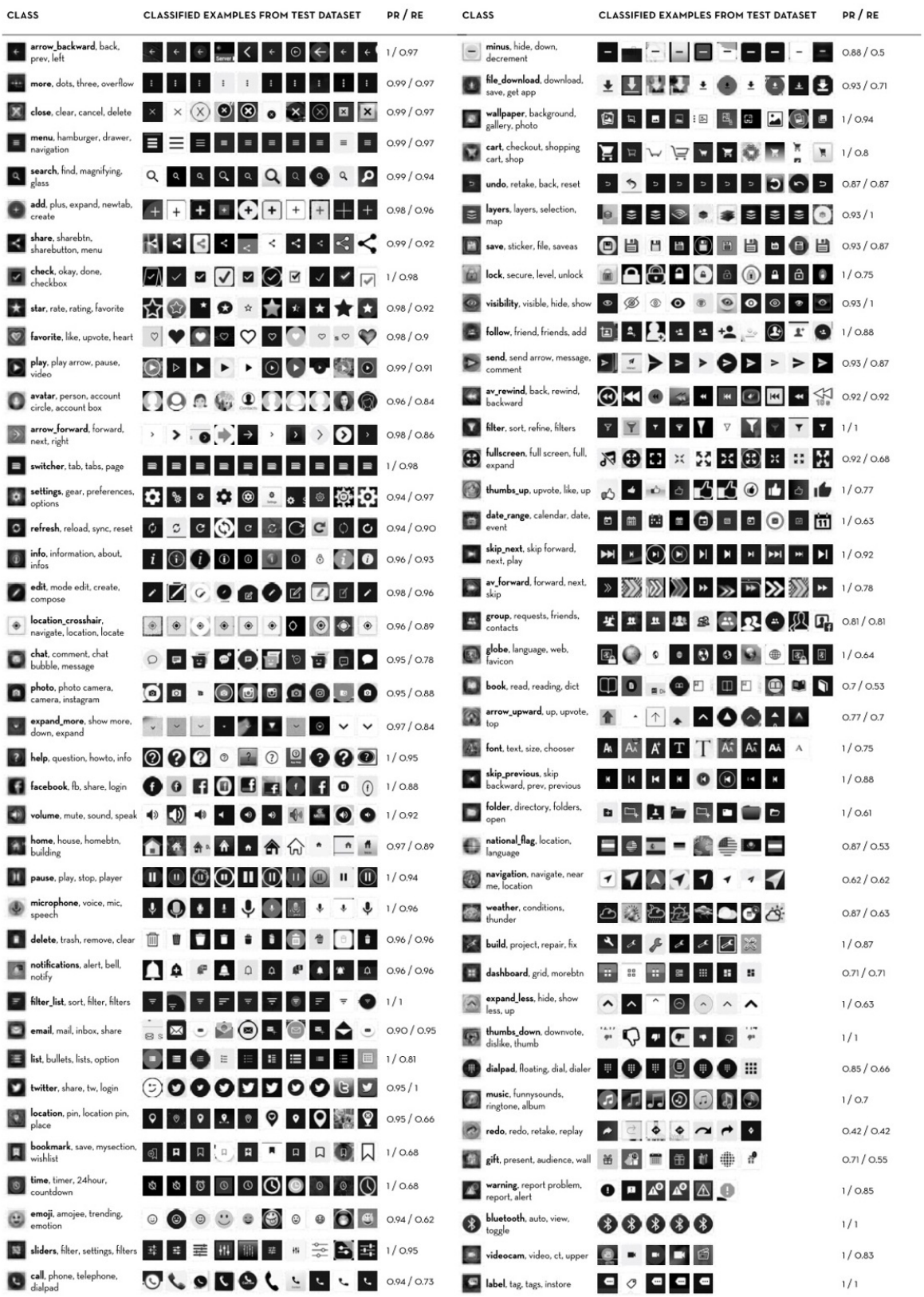

Figure 3.6: The 80 most frequent icon classes identified through an iterative open coding of 73k elements from the Rico dataset. Given the labeled data for each icon class, we compute an "average icon" and a set of related text concepts. For each icon class, we also present precision/recall metrics from a CNN trained to distinguish between 99 common categories, and 10 predicted examples from the test set.

CHAPTER 4: ICON FORMATIVE STUDY

To determine the usefulness of our icon dataset, we did a formative study where we attempted to create an embedding of mobile icons. We trained a small convolutional neural network on the icon dataset, clustered them, and used a nearest neighbors algorithm to evaluate the embedding.

4.1 EMBEDDING CNN ARCHITECTURE

The architecture of the embedding model was rather simple. The model consists of convolutions separated by max pooling and dropout layers, along with two fully connected layers at the end, a very boring architecture that's not particularly special [14].

Each convolutional layer used RELU activation and used padding to keep each layer the same size. The max pooling layers had filter size 2 and stride size 2. The dropout layers increased in dropout probability from 0.15 to 0.5. The final layer used softmax activation with a one hot encoding output. RMSprop was used for gradient descent to minimize the categorical cross entropy loss. An illustration of the model architecture can be seen in Figure 5.2. The result of the CNN on a classification task is shown in Figure 4.2.

4.2 CNN TRAINING

We trained the CNN on the new, larger dataset of images, with over 100 classes. The CNN is very lightweight, and only took 20 minutes to train 100 epochs on a 1080ti. As Figure 4.3 shows, the CNN converged quite nicely. Figure 5.3 displays some statistics for the CNN performance. Unfortunately, it seems that some classes are dragging down the macro precision and recall. Indeed, it turned out that the CNN was not classifying anything as 25 of the classes in training. Nevertheless, 85% accuracy is quite good for such a vanilla CNN.

4.3 CLUSTERING AND EVALUATION

To implement the idea of bringing icons with similar resource IDs closer together, pairwise constrained K-means clustering was used. Basically, if two resource IDs were similar enough, a constraint would be created between the two of them, penalizing the embedding for placing them further away. To generate the constraints, every icon resource ID was compared to every other resource ID, and a constraint created if they both shared a subword. For the sake of saving processing time and memory, we reduced the number of constraints to 10,000,000. Then, we ran

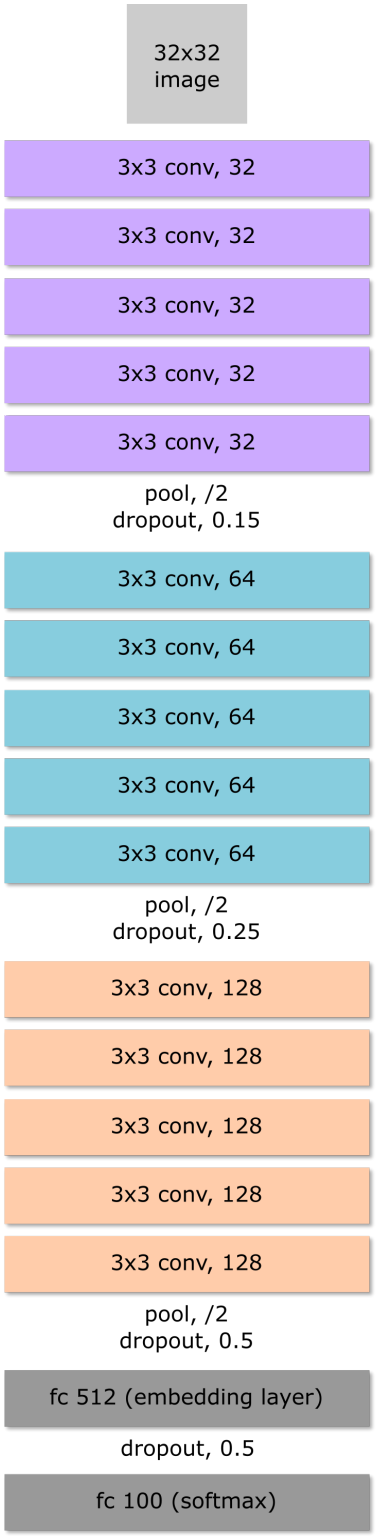

Figure 4.1: The 17 layer model architecture.

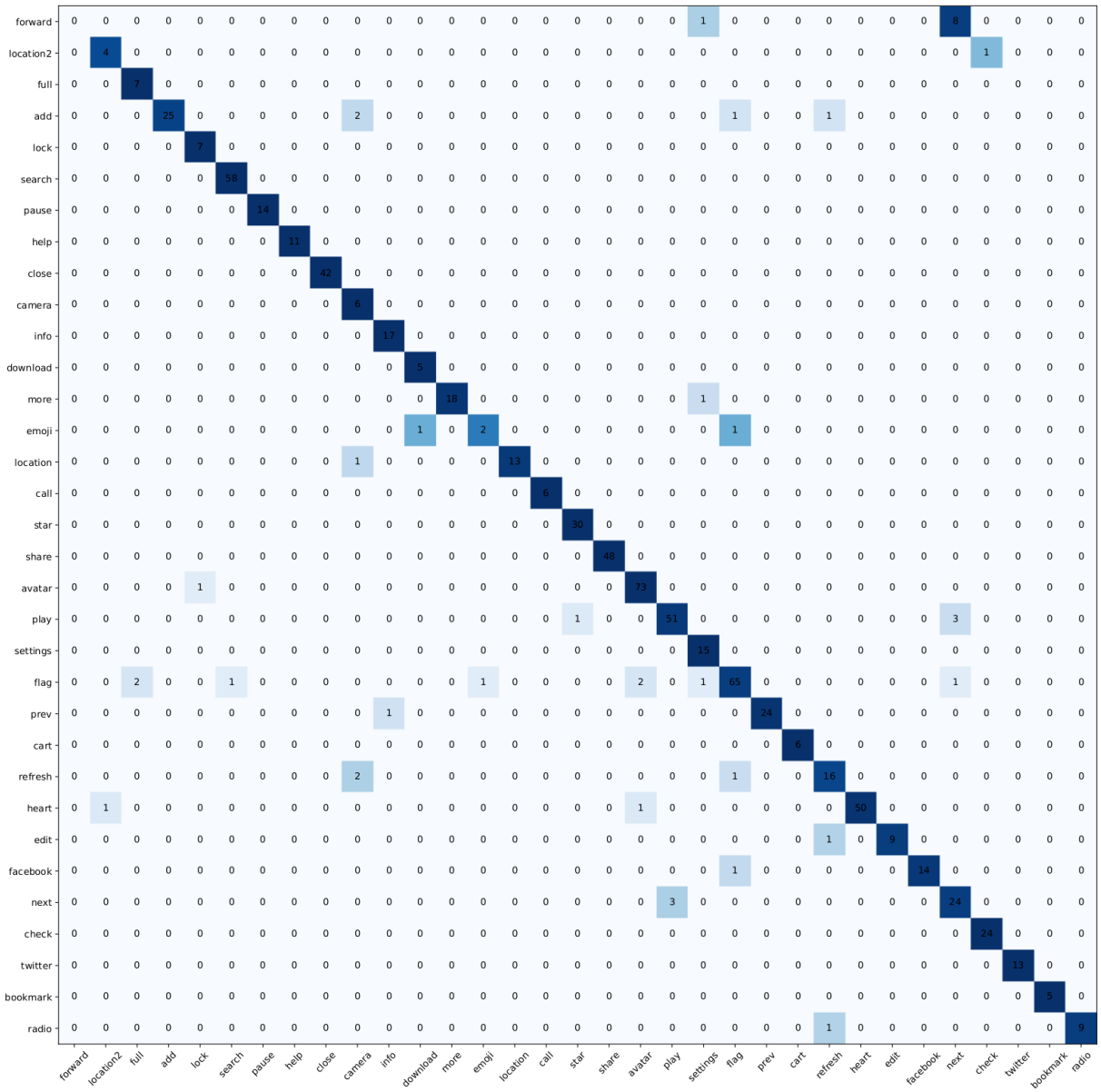

Figure 4.2: The testing confusion matrix of the small CNN.

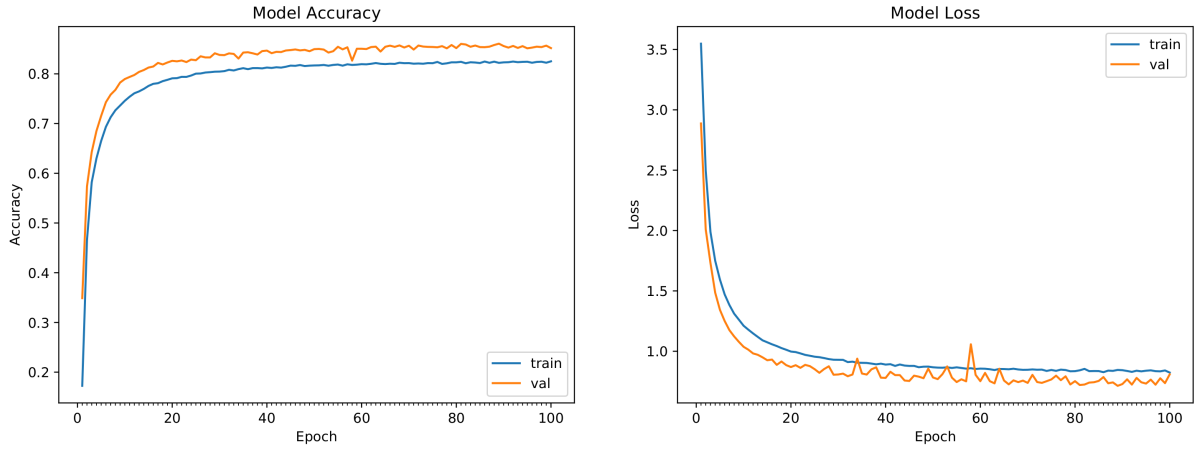

Figure 4.3: Training progress of the CNN over the full dataset.

Figure 4.4: Metrics for training and testing, respectively.

Accuracy	MacroP	MicroP	MacroR	MicroR
.8498	.4948	.8498	.4864	.8498

Accuracy	MacroP	MicroP	MacroR	MicroR
.8538	.4899	.8538	.4881	.8538

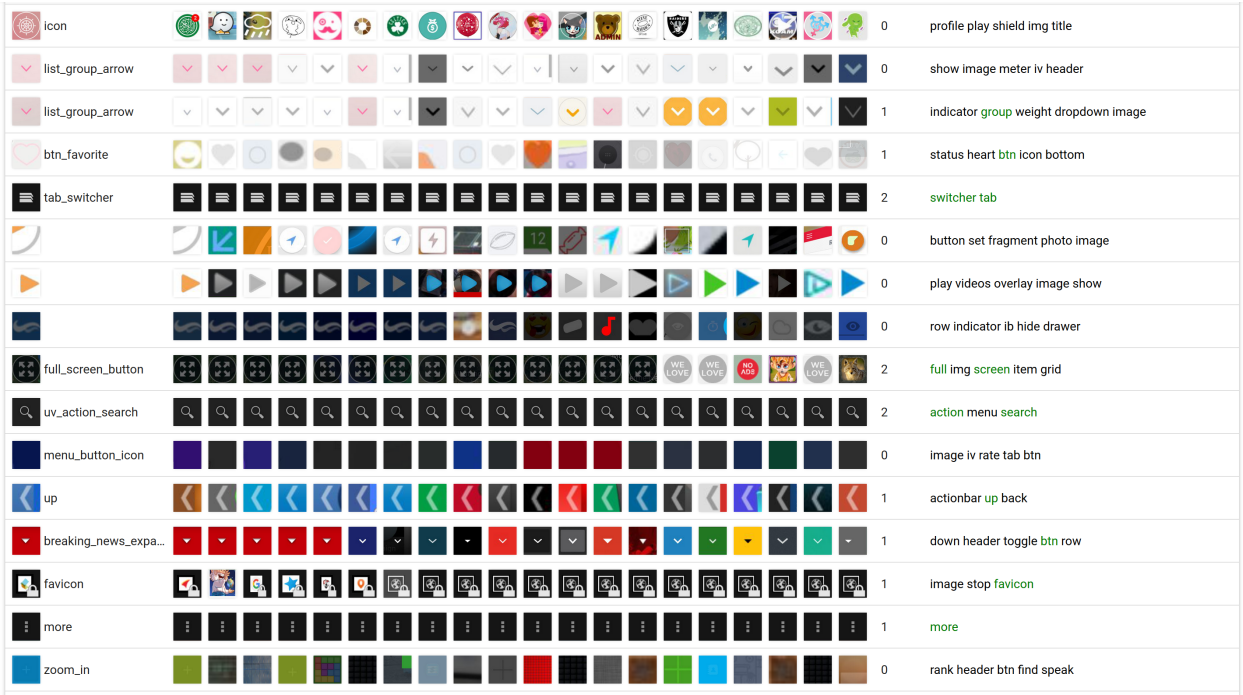

Figure 4.5: The web app displaying the results of the embedding.

the pairwise constrained K-means algorithm with 200 clusters. After the clustering was complete, the centroids were fed into the CNN and an updated embedding was created.

We then calculated the embeddings of the entire test set. Figure 4.5 displays a few examples of how the embedding worked. On the left, the icon to be labeled. In the middle, the twenty closest training icons. Finally, on the right, the most common "words" of the training icons; these operate as votes that label the icon on the left. Matches are highlighted in green to indicate that the voters have guessed a word correctly.

Interestingly enough, things like the "favicon" shown in Figure 4.5 were not a class in the original training set, but, demonstrating the robustness of the embedding, the neighboring favicons were able to label it successfully.

Less subjectively, we calculated a few metrics to evaluate the effectiveness of the embedding:

1. Accuracy: Whether at least one word in the top guesses of the neighboring icons matched the resource ID of the test icon. This is calculated only over icons that have a resource ID; not all programmers follow good practices and label their icons with resource IDs.

2. Mean Reciprocal Rank: Similar to accuracy, except it takes into account whether the "correct" word is more highly ranked than the "incorrect" one

The accuracy on the test set for this embedding task was 0.435 while the MRR was 0.237. This is a remarkable result given that the embedding task accuracy with the smaller CNN was only

0.221. Also, given that many of the resource IDs are completely uninformative or unintuitive, being able to label nearly half the icons correctly is a strong result in my book.

CHAPTER 5: AUTOMATING UI/UX DESIGN SEMANTICS

The second contribution of the thesis is an automatic approach for annotating mobile UIs with design semantics. Similar to how ImageNet [11] used WordNet to organize images, we leverage the lexical UI/UX database we create to understand mobile UI screens and semantically label their elements. The lexical database provides a UI/UX vocabulary, as well as labeled data that can be used to learn how to automatically detect different types of components and concepts. Given a UI's screenshot and view hierarchy, we are able to automatically identify 24 UI component categories, 197 text button concepts, and 99 classes of icons.

The automated approach iterates over all elements in a UI's view hierarchy, and uses code-based properties to identify different types of UI components and text button concepts. Since images and icons cannot be automatically differentiated through code, we trained a convolutional neural net (CNN) and anomaly detection pipleline to determine if an image belongs to one of 99 common classes of icons. We pass all visible image elements that are small and squarish through this pipeline. Using this automated approach, we computed semantic annotations for the 72k unique UIs comprising the Rico dataset, assigning labels for 78% of the total visible, non-redundant elements.

5.1 IDENTIFYING COMPONENTS

Using the labeled data provided by the lexical database, we discover and exploit similar underlying code-based patterns in the view hierarchy to semantically classify 23 out of the 24 types of UI components on a screen (Figure 3.4). To distinguish between icons and image content, we train a convolutional neural network and anomaly detection pipeline that we discuss later. For components lacking at least 30 example screens in the lexical database, we iterated through additional view hierarchies until each component had a support of 30 screens.

We identify many types of components (e.g., text buttons) using its `classname` or its `ancestors`' classnames, both properties readily available in the view hierarchy. For example, a Map View component contains the string "mapview" in its `classname`. To discover if characteristic classnames exist for a UI component category, we extract the `classname` and `ancestors` properties from elements in its category, and run a tf-idf analysis over the set of strings.

Other types of UI components are detected through structure-based patterns that examine an element's relation to its ancestors, descendants, or the view hierarchy itself. For example, certain types of UI components (e.g., "List Item") occur as children or descendants of specific types of other nodes that can be detected. Conversely, other types such as "Bottom Navigation" and "Pager

Indicator" can be determined by their children. Finally, "Modal" components can be identified as elements that occur at the top of the view hierarchy, but have an area that is smaller than the size of the phone screen.

To explain our heuristic in detail, we split up the 24 components into the following groups to simplify the explanation. Group A: Advertisement, Button Bar, Checkbox, Date Picker, Image, Image Button, Input, Map View, Multi-Tab, Number Stepper, On/Off Switch, Radio Button, Slider, Text, Toolbar, Video, Web View. Group B: Bottom Navigation, Drawer, List Item. Group C: Modal, Pager Indicator. Group D: Icon. Group E: Text Button.

Components in Group A: we mainly perform string comparison on each level of attributes. First check is on the class name. If the class name contains a substring of a keyword in our dictionary-class, we identified marked it as a component. If there are no class names matched in our dictionary, we checked on the ancestor's classes. If there is an ancestor's class that matched a keyword in our dictionary-ancestor, we identified a component. If the individual searches did not yield a component, we searched both the class name and an ancestor's classes simultaneously. In the case that none of the above comparisons applied, we repeated this process on all children, if they exist. Both class name and ancestor's classes contain an indication of a specific component. As mentioned before, Map View usually contains a substring "mapview" in its class name. Sometimes, a simple check on class name or ancestor's classes cannot determine a component. Therefore, we have to check on both the class name and the ancestor's classes together.

Components in Group B used a similar approach in Group A, except we labeled children elements as the identified components instead of the parent element. This is more accurate to check on the parent attributes than checking the children attributes. Parents components can sometimes show evidence of a component when children show no signs and do not refer a component because the programmer used a generic element. For Drawer and Bottom Navigation, we also checked whether the current level contains more than 2 children. These children are marked visible to the user in its element properties.

Group C is a special type of component. For each level, we checked on all four boundary values. If one boundary matched our discovered condition and children exist, the first child is the component we identified. Otherwise, we repeated this process on all children if they exist.

For icons in Group D, we first checked the size of the current level, whether it is too tiny and cannot be considered as an icon. The second check is on the widths and the heights percentage that occupied on the screen. An icon has a dimension ratio similar to the square shape. Percentage value indicates both the element ratio and the size, whether is is too big for an icon.

Text buttons, Group E, are described in section 3.3.1.

To measure the comprehensiveness of our component category set and automatic labeling approach, we computed the semantic coverage we could provide over the elements found in the Rico

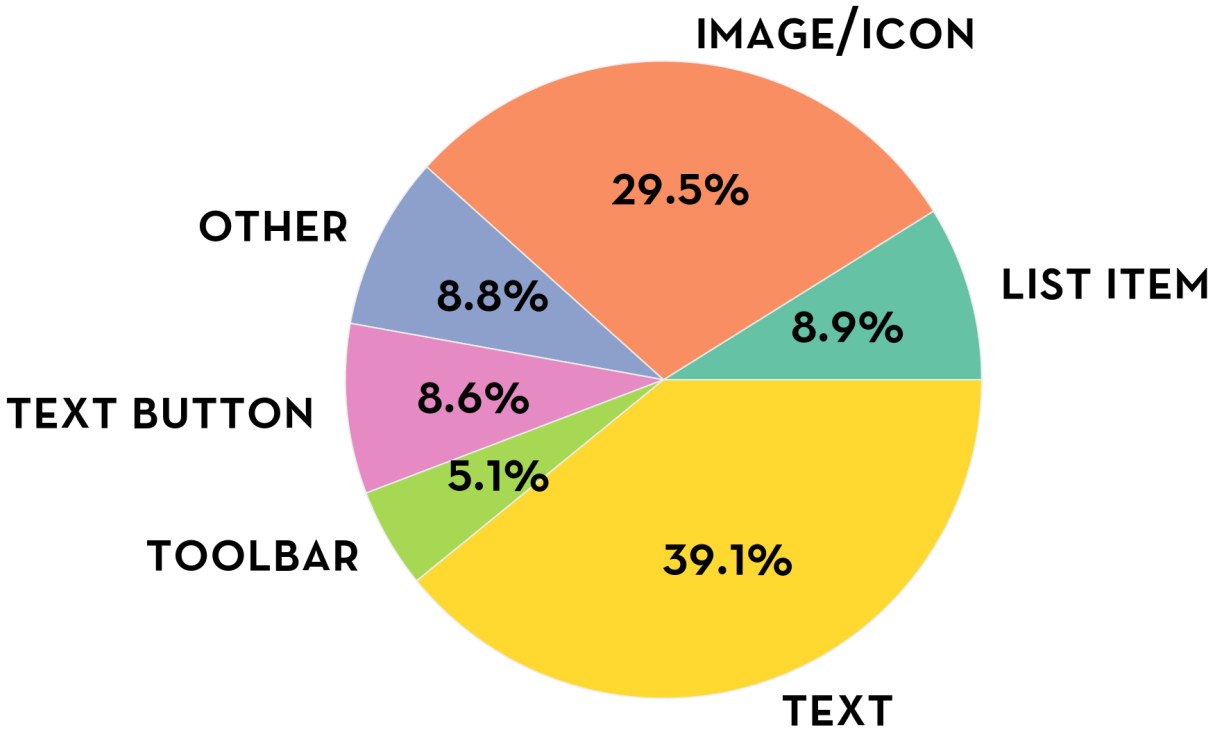

Figure 5.1: The distribution of the most common components we discovered within the screens we analyzed.

dataset. To compute the total number of elements in Rico which *could* have labels, we removed invisible, and redundant nodes that do not contribute to the UI screen [2]. In total, our approach can label 1,384,653 out of the 1,776,412 (77.95%) visible, non-redundant elements present in the Rico dataset, averaging 25 component annotations per UI. Figure 5.1 displays a more detailed breakdown of the component analysis. Text, images, text buttons, toolbars, and list items account for the vast majority of components.

5.2 CLASSIFYING ICONS

As explained earlier, icon classification is not a new problem. We borrow the approach that performed best on CIFAR-100 to use in this icon classification problem [10]. While icons tend to be simpler and more abstract than the images in CIFAR-100, we take their technique and apply it to this new domain.

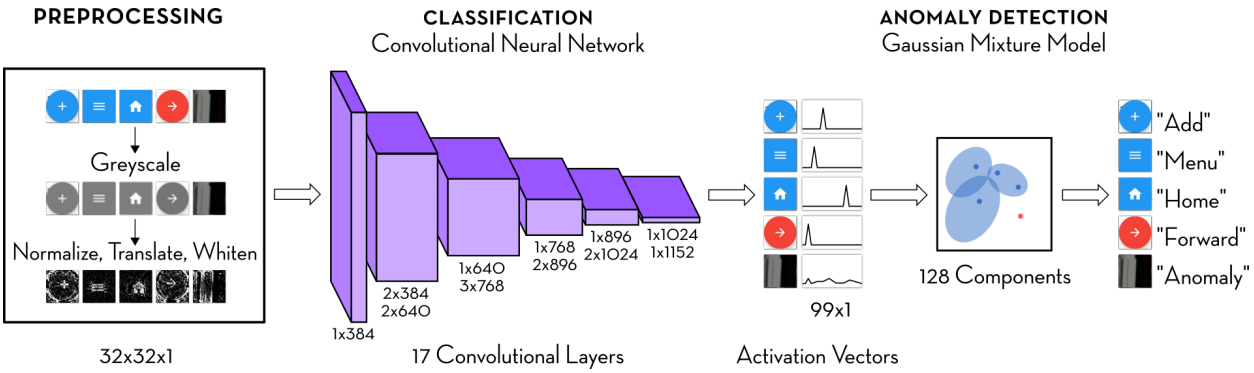

Figure 5.2: We pass all small, squarish images found in a UI's view hierarchy through a classification pipeline that distinguishes between 99 classes of icons and general image content using a CNN and anomaly detector, respectively. CNN stack labels denote (layers $\times$ filters).

5.2.1 Preprocessing

To create the icon dataset for training the convolutional neural network, we use the icons we extracted and labeled. Before creating a cutoff for classification, classes with the same meaning or interchangeable appearance were combined to decrease ambiguity and increase the support for merged classes. For instance, the *more_horizontal* and *more_vertical* classes were combined into a single *more* class, since one is just a rotation of the other; *warning* , *error* , and *announcement* are all visually similar icons used interchangeably to call attention to something in a user interface and were combined as well. We only used icon classes with more than 40 examples for training. In total, 99 classes had over 40 examples of icons from the dataset. Additionally, a total of 49,136 icons were labeled as fitting into one of the labels in our list of classes.

We found that a few preprocessing steps improved the accuracy of the CNN. First, we converted icons to greyscale to simplify operating on the icons and eliminate the effect of color on the convolutional neural network, since color is usually not a factor in the design and comprehensibility of an icon [25, 26]. Using greyscale images instead of RGB images not only improved accuracy, it also increased the speed of the CNN by reducing the dimensionality of the problem.

Several types of additional data preprocessing and augmentation were helpful.

1. Featurewise centering and featurewise standard deviation: the mean over the entire dataset was set to 0 and the standard deviation to 1. Samplewise centering and samplewise standard deviation normalization worsened the result and were not used.

2. Translation: images were randomly moved along the x and y axis by up to 10% of the image width.

3. ZCA whitening: ZCA whitening improved results, likely because icon meaning relies more

on its shape and symbolism than its color, fine details, or stylization [26].

5.2.2 Model Architecture

In choosing the CNN architecture to use on our dataset, we used the network that currently has the best results on CIFAR-100 [10], since CIFAR-100 is similar to our problem in the number of the classes and the size of the input. This model comprises 17 convolutional layers, arranged into 6 stacks. After each stack, we applied max pooling with a stride and filter size of 2. Increasing dropout of (0.0, 0.1, 0.2, 0.3, 0.4, 0.5) followed each stack as well. At the end, we added two fully connected layers, one with 512 neurons and a final softmax activation layer with 99 neurons. We trained the network to minimize categorical cross entropy loss with RMSprop. All layers except the final one used ELU activation.

5.2.3 CNN Results

We trained the CNN on the full dataset of icons, with 99 classes. Like before, 10% of the data was set aside for testing.

Figure 5.3 displays some statistics for the CNN performance. With an accuracy of 99.49% in train and 94.43% in test, the performance of the CNN is state of the art compared to results on CIFAR-100 [10]. Our CNN achieves substantially higher accuracy because there is much less variance within classes of icons, making the classification task easier compared to CIFAR-100. The macro precision and macro recall metrics are the averages of precision and recall of each class, calculated by simply summing individually calculated metrics for each class and dividing by 99. Unlike accuracy, each class is weighed equally in these two metrics, demonstrating that the CNN can robustly classify icons.

	NO ANOMALY DETECTION		WITH ANOMALY DETECTION	
	TRAIN	TEST	TRAIN	TEST
ACCURACY	0.9949	0.9443	0.9712	0.9098
MACRO PRECISION	0.9946	0.8743	0.9783	0.9062
MACRO RECALL	0.9946	0.8573	0.9697	0.7910

Figure 5.3: Icon classification performance metrics for 99 common classes, with and without anomaly detection.

	NORMAL	ANOMALY
PRECISION	0.90	0.94
RECALL	0.98	0.78
F1-SCORE	0.94	0.85
SUPPORT	48829	23906

Figure 5.4: The accuracy metrics for anomaly detection.

5.2.4 Anomaly Detection

Because the CNN was only trained on valid icons, when the CNN classifies arbitrary elements from UIs, it needs a way to detect anomalous images that it cannot properly classify, i.e. images that differ significantly from its training set. We took the approach of looking for anomalous activations of the CNN's last softmax layer. We generated all the activations of the training set, and then trained a Gaussian Mixture Model on these activations. Specifically, we used scikit-learn's GaussianMixture with 128 components and full covariance [27]. Figure 5.2 shows the icon classification pipeline in its entirety.

Viewing anomaly detection as a binary classification problem, we were able to achieve a recall score of 0.98 on valid icons, and a precision of 0.94 on detecting anomalous icons, indicating that we were able to discard anomalies while only discarding 2% of valid icons. We achieved a precision score of 0.9 on valid icons, meaning that only a small percentage of anomalies are classified as valid. Figure 5.4 shows the full range of measured with which we evaluated the anomaly detection.

The accuracy, precision, and recall metrics of the CNN combined with the anomaly detection GMM are displayed in Figure 5.3. While the accuracy of the classifier does decrease when anomaly detection is added, the decrease in accuracy is less than 5%. Notably, the precision of the CNN *increases* slightly while its recall decreases. This is because the anomaly detector removes some valid icons that appear anomalous, i.e. the icons that the CNN is less sure about and therefore "harder" to place into the right class, increasing its precision. However, because valid icons are thrown out as anomalies, recall decreases.

We use the code-based patterns for detecting UI components and the icon classification pipeline

to semantically annotate the 72k unique UIs in the Rico dataset. We use code-based rules to recognize all categories of elements except for icons. To identify icons, we pass all small, squarish images found in a UI's view hierarchy through the classification and anomaly detection pipeline. Once we semantically label the elements in a view hierarchy, we also generate a semantic version of its screenshot, which are useful representations for vision-based data-driven applications (Figure 3.4).

Because the icon classifier can identify icons that it cannot classify via anomaly detection, we liberally treated many images as potential icons in order to cast as wide a net as possible. Then, we used the individual rules we came up with for each of the 24 components to identify the components for all of the UI screens from the RICO dataset. After identifying each of the components, we use the bounding boxes from the view hierarchies to color the components as seen in Figure 3.4.

5.2.5 Using CNNs for Icon Text Synonym Evaluation

To evaluate the usefulness of the icon text synonyms we mined earlier, a few of which are shown in Figure 5.5, we used it in a classification task. Because the CNN model architecture described in Figure 5.2 already performed very well, we used a simpler CNN to test the utility of the icon text synonyms. This CNN had three stacks of convolutional layers arranged in ($5 \times 32 \times 3$, $5 \times 64 \times 3$, $5 \times 128 \times 3$) layers $\times$ units $\times$ receptive fields. It achieved an accuracy of of .9291, a macro precision of 0.8212, and a macro recall of 0.859 on the icon test set. We then encoded each icon's resource ID as a simple binary vector indicating the synonym words found in the resource ID. We augmented the CNN with this additional information by concatenating the first fully connected layer of the CNN with this binary vector. The simple CNN achieved an accuracy of .9466, a macro precision of .8745, and a macro recall of 0.8634 on the test set, which is comparable to the more complex network. The increase in classification metrics across the board indicates that the synonym set provides information useful enough for machine learning tasks.

location	pin location pin | location place map spec locator waypoint poi stores tab2 nearby
list	bullets | lists list lang shopping map location option alerting switch nav
bookmark	save | bookmark bookmarks mysection save teaser soon try wishlist ribbon favorite
delete	trash remove | delete clear remove del trash history ruledout uninstall edit erase
emoji	| emoji amojee trending emotion emoticon minutiae mood face happy sticker
minus	hide | minus out zoom down bi decrement decrease quantity plus remove
microphone	voice mic | voice mic speech ex search record or microphone talk translate
notifications	alert | notification notifications id bell notify alerts reminder score alarm notice
twitter	share | twitter switcher tab share tw login imgtwitter social sign it
access_time	time | timer 24hour countdown time ago recent clock just 15mins count
call	phone telephone | call dialpad whatsapp floating phone parmbt contact advert dial us
email	mail inbox | email mail share contact send ringtone emaill gmail msg inbox
cart	checkout cart shopping cart | cart shop buy shopping basket tap compra store buttom area
filter_list	| sort filter filters actionbar by heart groupoption menu score action

Figure 5.5: A sampling of some of the synonyms extracted from the data.

CHAPTER 6: DESIGN APPLICATIONS

Finally, we sketch how semantics can power new types of data-driven design interactions, including a UI/UX design reference, design search over screens and flows, and generative tools for design.

6.1 UI/UX DESIGN REFERENCE

The lexical database itself can be used as a design resource to understand how UI components and UX concepts map to implementations in existing apps. For each category of UI component, the lexical database provides examples from the Rico dataset. For each UX concept, the lexical database reveals button text and visually distinct icons related to the concept (Figure 6.1). This mapping reveals if concepts are most often expressed as text or icons, and if there are visually distinct icons that have similar meaning (e.g., *add* ⊞ and *edit* ◪). We found that 94 UX concepts are expressed both as text buttons and as icons. Whereas, some concepts such as *navigation* ▲ and *gift* 🎁 are solely graphical ones.

6.2 SEMANTIC SEARCH OVER SCREENS AND FLOWS

Prior work has shown the utility of semantics in mobile design search applications [3, 4]. Deka et al. demonstrated how training an embedding using UI screens that encode image and text content could power example-based UI searches that could often recover visually and *semantically* similar UI screens [4]. We implement a similar application using the semantic versions of the UI screenshots we computed over the Rico dataset.

We trained a convolutional autoencoder, downsampling the images to 512×256. The encoder half of the autoencoder consists of 6 convolutional layers arranged as (8×3, 8×3, 8×3, 4×3, 4×3, 4×3) filters $\times$ receptive fields. A max pooling layer of size and stride 2 is applied after every convolutional layer, resulting in a 128 length vector as the encoded representation. The decoding half of the autoencoder is simply the encoding half in reverse order, with upsampling layers instead of max pooling layers, and a final 3×3 layer to convert back to the original RGB input. Once the autoencoder converged, we used it to embed all the UIs in the Rico dataset in the learned semantic space, and then inserted the embedded vectors into a KD-tree to run fast nearest neighbor searches (Figure 6.2).

We can also use our design semantics for analyzing user flows. A user flow is a group of UI screens that are part of accomplishing a specific task in an app. Previous research has shown

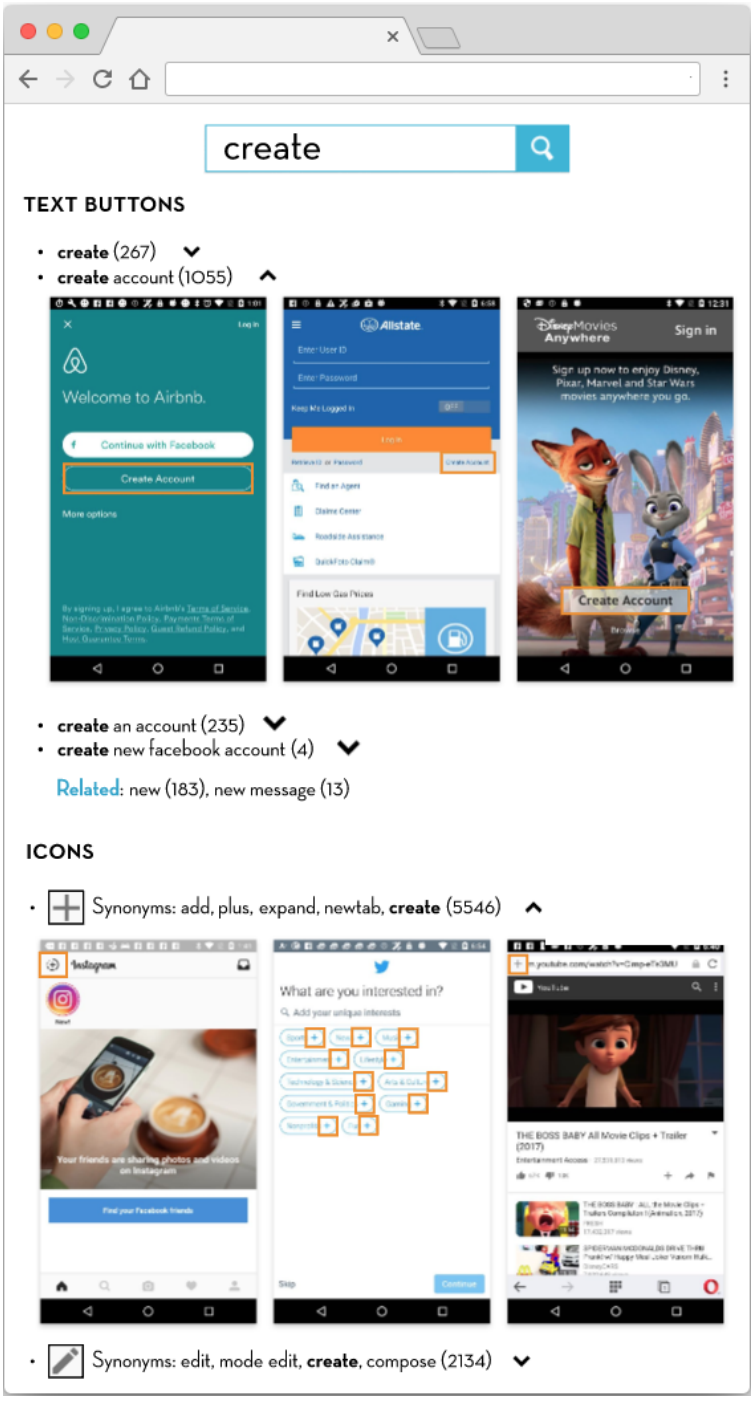

Figure 6.1: The lexical database can be used as a design resource to understand how UX concepts map to text buttons and icons, exposing if there are multiple textual and graphical instantiations of the same concept (e.g., *add* ⊞ and *edit* ⊠). For each icon, it also provides a synonym set of text concepts, revealing the icon's range of meaning.

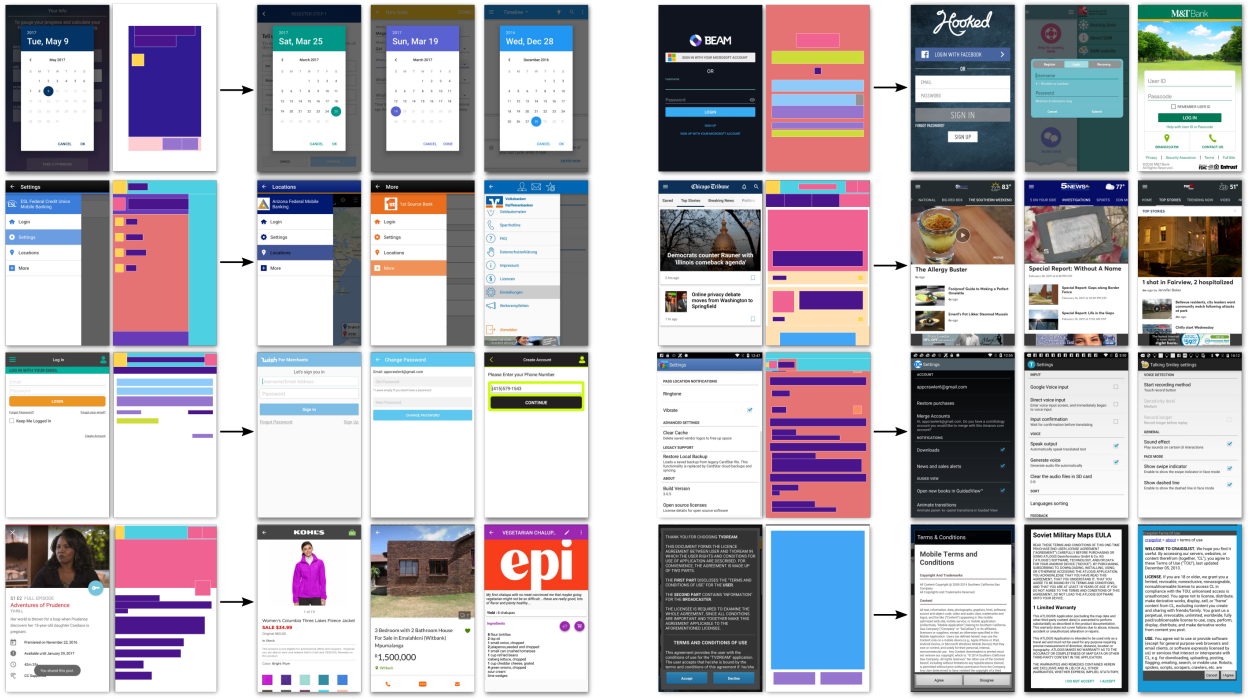

Figure 6.2: Taking as input the semantic versions of screenshots that we computed, we trained a deep convolutional autoencoder to embed Rico's unique UI dataset. We inserted the embedded vectors into a KD-tree to enable fast, example-based semantic searches over UI screens.

that user flows can be identified by examining a UI element that was clicked on a screen [3]. By identifying text buttons and icons that are commonly used to change app state and the most common icons, we enable a similar analysis. We identified 14,495 clicked interactions in Rico's 72k dataset of unique UIs, of which 9,461 were buttons or images. Out of the clicked buttons and images found, we were able to identify 57.5% of the elements clicked as either a recognized text button or a non-anomaly icon.

6.3 TOOLS FOR GENERATING DESIGNS

Semantics are also useful for building design tools that support generation. As mentioned earlier, several approaches including REDRAW, REMAUI, and P2A have been developed for automatic UI generation [13, 28, 16]. By taking as input semantically labeled components, more powerful probabilistic generative models of mobile UIs can be learned. Techniques such as generative adversarial networks and variational autoencoders [29] can be used to build design tools that can "autocomplete" partial specifications of designs. For example, the tool can recognize that a designer is creating a "login" screen based on the central placement of a login button, and suggest adding other elements such as "username" and "password" input fields. These tools can translate a

set of semantic constraints specified by a designer (e.g., login button) into a cohesive visual layout and information architecture, as well as suggest UI elements that are missing from the design.

REDRAW, the most advanced of the three tools, explains that its vision based approach can often be confused by visually similar but functionally differing components. By using the trace data from real user interactions that RICO provides, it could be possible to more reliably produce user interface code. Additionally, these tools focus on generating UI code from high fidelity mockups produced by tools like Photoshop, which come in the later stages of design. A component, button, and icon model based approach would allow for more robust generation of entire app flows, as well as suggestions for alternative UI designs. In the earlier ideation or brainstorming stages of design, iterating through a large number of design variations is preferred to creating high quality mockups with advanced tools [30].

CHAPTER 7: LIMITATIONS AND FUTURE WORK

In addition to using design semantics to bootstrap data-driven interactions, future work should focus on broadening the capabilities of the icon classifier by improved training on icons with fewer examples. Further research can also focus on generalizing the design semantics we present to other mobile platforms.

7.1 IMPROVED CNN PERFORMANCE ON THE LONG TAIL

Although our convolutional network had exceptional performance on the icon classes that had hundreds or thousands of training examples, its performance suffered on icons with fewer examples in the dataset. Additional augmentation of the dataset is a clear avenue of improving the CNN's robustness on icons. Indeed, the initial scraping of icons from the Rico dataset used a somewhat conservative selection method. Further work in labeling a larger portion of icons would undoubtedly improve performance.

However, research on one-shot image recognition could also be exploited to allow for the detection of icons with low support, without the need to undertake a manual labeling effort. Siamese neural networks have shown promise on the MNIST dataset of handwritten characters [31]. Given the similar size of MNIST images to icons, a similar approach could prove effective, especially given the prior knowledge learned from icons with larger support.

7.2 BUILDING CROSS PLATFORM CLASSIFIERS

We only analyze Android apps and UIs for the identification of UI components, text buttons, and icons. However, applying the techniques we used is largely an engineering problem. For iOS, we could use Apple's UI testing framework (XCUITest) to take screenshots and capture the view hierarchy of app screens. Afterward, we could analyze the iOS apps in the same way as we did for Android.

The mobile UI components we found present a more complex challenge for cross platform application. Our approach relies partially on identifying Android classes. To enable a unified cross platform approach to UI components, the collection of UI components identified using our Android based rules could be cropped from the corresponding screenshots to create a training dataset. As with icons, this training dataset could be used to train a vision based classifier, allowing cross platform application by simply examining user interfaces visually. A vision based approach can also help with identifying outlier components that are dynamically created through Android Web

Views, whose contents are not visible in the view hierarchy [16].

REFERENCES

[1] K. Alharbi and T. Yeh, "Collect, decompile, extract, stats, and diff: Mining design pattern changes in android apps," in *Proc. MobileHCI*, 2015, pp. 515–524.

[2] A. Sahami Shirazi, N. Henze, A. Schmidt, R. Goldberg, B. Schmidt, and H. Schmauder, "Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps," in *Proc. SIGCHI*, 2013, pp. 275–284.

[3] B. Deka, Z. Huang, and R. Kumar, "Erica: Interaction mining mobile apps," in *Proc. UIST*, 2016, pp. 767–776.

[4] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proc. UIST*, 2017, pp. 845–854.

[5] A. Sulek, "Collectui," 2016. [Online]. Available: http://collectui.com/designs

[6] M. Sheibly, "Mobile patterns," 2013. [Online]. Available: http://www.mobile-patterns.com/

[7] L. Yi, L. J. Guibas, A. Hertzmann, V. G. Kim, H. Su, and E. Yumer, "Learning hierarchical shape segmentation and labeling from online repositories," *CoRR*, vol. abs/1705.01661, 2017.

[8] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[9] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[10] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *CoRR*, vol. abs/1511.07289, 2015.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. CVPR*, 2009, pp. 248–255.

[12] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, pp. 39–41, 1995.

[13] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," *arXiv preprint arXiv:1705.07962*, 2017.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[15] S. Natarajan and C. Csallner, "P2a: A tool for converting pixels to animated mobile application user interfaces," p. 12, 2018.

[16] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," *arXiv preprint arXiv:1802.02312*, 2018.

[17] I. MonboDB, “Mongodb,” 2018. [Online]. Available: https://www.mongodb.com/

[18] closeio, “Flask-mongorest,” September 2017.

[19] F. Inc., “React,” 2018. [Online]. Available: https://reactjs.org/

[20] B. Studios, “basalmiq,” 2018. [Online]. Available: https://balsamiq.com/

[21] Call-Em-All, “Material-ui,” 2018. [Online]. Available: https://material-ui-next.com/

[22] E. Boling, J. E. Beriswill, R. Xaver, C. Hebb et al., “Text labels for hypertext navigation buttons,” *International Journal of Instructional Media*, vol. 25, p. 407, 1998.

[23] Google, “Material icons,” 2017. [Online]. Available: https://material.io/icons/

[24] Y. Rogers, “Icons at the interface: their usefulness,” *Interacting with Computers*, vol. 1, pp. 105–117, 1989.

[25] C. J. Kacmar and J. M. Carey, “Assessing the usability of icons in user interfaces,” *Behaviour & Information Technology*, vol. 10, pp. 443–457, 1991.

[26] C.-C. Chen, “User recognition and preference of app icon stylization design on the smartphone,” in *Proc. HCII*. Springer, 2015, pp. 9–15.

[27] scikit-learn developers, “Gaussian mixture models,” 2017. [Online]. Available: http://scikit-learn.org/stable/modules/mixture.html

[28] T. A. Nguyen and C. Csallner, “Reverse engineering mobile application user interfaces with remaui (t),” in *Proc. Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, 2015, pp. 248–259.

[29] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Proc. NIPS*, 2014, pp. 2672–2680.

[30] J. A. Landay and B. A. Myers, “Interactive sketching for the early stages of user interface design,” in *Proc. SIGCHI*, 1995, pp. 43–50.

[31] G. Koch, R. Zemel, and R. Salakhutdinov, “Siamese neural networks for one-shot image recognition,” in *Proc. ICML Deep Learning Workshop*, vol. 2, 2015.